Turning Futexes Inside-Out: Efficient and Deterministic User Space Synchronization Primitives for Real-Time Systems with IPCP

Alexander Zuepke
RheinMain University of Applied Sciences, Wiesbaden, Germany
alexander.zuepke@hs-rm.de

Abstract

In Linux and other operating systems, futexes (fast user space mutexes) are the underlying synchronization primitives to implement POSIX synchronization mechanisms, such as blocking mutexes, condition variables, and semaphores. Futexes allow one to implement mutexes with excellent performance by avoiding system calls in the fast path. However, futexes are fundamentally limited to synchronization mechanisms that are expressible as atomic operations on 32-bit variables. At operating system kernel level, futex implementations require complex mechanisms to look up internal wait queues making them susceptible to determinism issues. In this paper, we present an alternative design for futexes by completely moving the complexity of wait queue management from the operating system kernel into user space, i.e. we turn futexes "inside out". The enabling mechanisms for "inside-out futexes" are an efficient implementation of the immediate priority ceiling protocol (IPCP) to achieve non-preemptive critical sections in user space, spinlocks for mutual exclusion, and interwoven services to suspend or wake up threads. The design allows us to implement common thread synchronization mechanisms in user space and to move determinism concerns out of the kernel while keeping the performance properties of futexes. The presented approach is suitable for multi-processor real-time systems with partitioned fixed-priority (P-FP) scheduling on each processor. We evaluate the approach with an implementation for mutexes and condition variables in a real-time operating system (RTOS). Experimental results on 32-bit ARM platforms show that the approach is feasible, and overheads are driven by low-level synchronization primitives.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Software and its engineering → Mutual exclusion

Keywords and phrases Futex, Immediate Priority Ceiling Protocol, Critical Section, Monitor

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.11

1 Introduction and Motivation

A common technique to improve the performance of synchronization mechanisms is to avoid unnecessary system calls [3, 4, 14, 18, 31]. Ousterhout [27] observed for processor architectures of the 1980s that "*[. . .] operating system performance does not seem to be improving at the same rate as the base speed of the underlying hardware*". Today, the situation has not changed much: system calls are an order of magnitude slower than atomic operations. For current Intel CPUs, Al Bahra measures about 15 cycles for an atomic *compare-and-swap (CAS)* operation on an Intel Core i7-3615QM [1], while Soares and Stumm describe a system call overhead of around 150 cycles on an earlier Core i7 generation [33]. For the future, we can assume that system calls remain expensive due to pipeline flushes [33] and mitigation against processor design flaws such as *Meltdown* [21] and *Spectre* [19]. While we can assume that processor design flaws like Meltdown will be fixed in future processor generations, Spectre-like attacks on branch prediction by data cache side channels are expected to stay, along with the corresponding measures of mitigation [23]. Therefore, avoiding unnecessary system calls is still relevant today for efficient synchronization mechanism.


© Alexander Zuepke;
licensed under Creative Commons License CC-BY

Futexes were originally conceived as a mechanism for *fast mutexes* in Linux [14]. The idea behind futexes is to acquire and release uncontended mutexes by just using atomic operations in the C library in user space. The kernel is only involved in handling blocking and unblocking of threads on contention. This design omits system calls into the operating system kernel in the fast path. Today, futexes are a generic *compare-and-block* mechanism in Linux and used to implement almost all blocking POSIX thread synchronization primitives. A further benefit of futexes is that no pre-registration of synchronization objects in the kernel is needed. The Linux kernel creates in-kernel objects such as wait queues on demand.

From the design point of a real-time system, the basic idea of futexes as an *optimization of the common case* also helps in mixed-criticality environments or when trying to improve *size, weight and power* (SWaP) considerations. If a real-time task finishes early, it is a desirable design goal to leave the remaining processing time to other non-real-time tasks, or to enable power saving strategies earlier. This principle applies to all kinds of optimizations such as processor caches, but only as long as an optimization also shows a deterministic behavior and one can analytically derive a proper *worst-case execution time (WCET)*.

However, design aspects that are important for the flexibility of futexes in Linux also bring problems for their deterministic behavior. In the Linux kernel, blocked threads are kept in a fixed-sized hash table, and previous research has shown that the futex hash table is vulnerable to cause interference to otherwise unrelated processes due to inadvertent or intentional hash collisions [38].

Alternative approaches to prevent this problem of *undesired resource sharing* involve pre-registration of synchronization objects. This allows indexing of wait queues by other means not susceptible to attacks, e.g. by using an index in a descriptor table [34]. Such a limitation is fully compliant to most programming environments for real-time applications, such as ARINC 653 for Avionics, which often require allocation of all resources at start time anyway, to prevent later resource exhaustion at runtime, but this also limits the flexibility of futexes for non-real-time workloads.

Another important design aspect of futexes is the use of 32-bit variables as protocol variables in user space. 32-bit variables were originally chosen because atomic operations on them are available on most major platforms supported by Linux. However, using 32-bit variables in futex-based synchronization protocols requires that all data needed to make a decision whether to wait or to wake up threads must be "compressed" into a single 32-bit variable. While this is possible for most POSIX user space synchronization mechanisms such as mutexes, condition variables, barriers, and semaphores, it is not the case for other, more complex synchronization mechanisms, such as message queues.

To address these problems of futexes in the context of real-time systems, while preserving their desirable properties, we propose to turn the futexes "inside-out" and to move the wait queue management *out* of the kernel *into* user space. An efficient mechanism for *short* critical sections is the enabling factor for queue manipulations, e.g. priority-ordered queues, which cannot be realized as simple atomic operations on modern hardware. Such a mechanism also removes the limitation of 32-bit protocol variables, allowing more complex waiting conditions. For this to work *efficiently* and *deterministically*, we need three key components:

(i) A reliable and efficient mechanism to prevent preemption in user space.

(ii) Fair spinlocks for mutual exclusion between other processor cores.

(iii) A light-weight mechanism to suspend and wake up threads.

The resulting design effectively resembles a *non-preemptive busy-waiting monitor* with *Mesa-style blocking condition variables* [10]. The main challenge of this design is that the overall performance of these mechanisms must be comparable to futexes. For a blocking mutex, we aim for a solution that avoids system calls in the fast path and requires system calls only

to suspend the calling thread in a mutex lock operation, or to wake up a waiting thread in an unlock operation. The baseline is given by a traditional approach of implementing synchronization mechanisms in the kernel, using dedicated system calls for all operations. Here, we must reach a similar level of performance in the worst case.

Components (i) and (ii) are well researched in the context of real-time systems. Specifically, in user space, temporarily disabling preemption while holding a spinlock is necessary to avoid the lock-holder preemption problem [26, 35]. To this end, efficient and predictable implementations of the *immediate priority ceiling protocol* (IPCP) [2, 37] are to be preferred over other non-real-time techniques of controlling preemption [13, 20, 22, 25]. In this paper, the design of component (iii) is based on the key idea of addressing threads directly. Mechanisms for $\mathcal{O}(1)$ look-up of threads by their ID are in fact readily available at OS kernel level, so we can leverage them to construct light-weight suspension and wake-up mechanisms targeting threads directly rather than introducing another level of indirection to look-up wait queues in the kernel. As result, the required kernel mechanisms to support the proposed monitors in user space are much simpler compared to futexes or a traditional approach. This effectively reduces the implementation effort and related WCET concerns inside the operating system kernel and moves the complexity to support common thread synchronization mechanisms into user space.

Our contributions in this paper are:

- A mechanism to suspend and to wake up threads in IPCP for an operating system using partitioned fixed-priority (P-PF) scheduling (Section 4).
- Non-preemptive busy-waiting monitors in user space (Section 5).
- An analysis of synchronization mechanisms w.r.t. their potential for optimization by reducing system calls and related WCET concerns (Section 6).
- An efficient implementation of blocking mutexes and condition variables based on the monitor that require at most one system call. Like futex-based synchronization mechanisms, no pre-registration of synchronization objects in the kernel is required (Section 7).
- An experimental evaluation of the approach in a research RTOS for 32-bit ARM processors to demonstrate the effectiveness of the proposed approach (Section 8).

2 Terminology and System Model

We assume a system comprising a contemporary 32-bit or 64-bit processor with one or more processor cores. On the system, an operating system kernel executes in *supervisor mode*, which is a privileged mode of the processor, while processes host code executing in a non-privileged *user mode*. The processor provides virtual memory or enforces memory protection, such that each process has its own isolated address space. Code in user mode uses *system calls*, a hardware trap mechanism, to call into the operating system kernel. The processor also provides general atomic operation on 32-bit variables based on *compare-and-swap (CAS)* or *load-linked/store-conditional (LL/SC)* instructions. The atomic operations additionally provide *acquire* or *release* semantics, or the processor provides explicit memory barriers to order memory accesses. We also assume that, on average, non-atomic operations (e.g. ALU- and normal load/store-operations) are much faster than atomic operations, and atomic operations are much faster than system calls.

We use the term *thread* instead of *task* when talking about schedulable entities of a process. *TCB* refers to the *thread control block*, which can comprise variables shared between kernel and user space. We denote the user space part of the TCB as *UTCB*. The identifier `SELF` points to the current thread's thread-local storage (TLS) data segment in user space, which also contains the UTCB.

We assume *partitioned fixed-priority* (P-FP) scheduling. We define the *scheduling priority* in the sense that higher values refer to higher priority level, as this is often done in RTOS implementations. The value `max_prio` defines the maximum priority level that a thread in a process can use, while `0` is the minimum priority level in the system.

For resource sharing on each single processor core, we use the *immediate priority ceiling protocol* (IPCP) [12]. When accessing a shared resource, the requesting thread temporarily raises its scheduling priority to the *ceiling priority* of the resource. The ceiling priority of each resource is defined as the maximum priority of all threads accessing the resource. This effectively excludes other threads from accessing the resource at the same time, but only as long as a thread does not block. Compared to the *original priority ceiling protocol* (OPCP) [32], IPCP is simpler to implement, and it is the implementation mandated by e.g. POSIX and found in real-world RTOS implementations. When `max_prio` is used as ceiling priority, the protocol effectively reduces to *non-preemptive critical sections* (NPCS).

We now consider resource sharing between two or more processor cores. For *short* critical sections, we combine non-preemptive critical sections with FIFO spinlocks, like in MSRP [15]. As spinlock implementation, both *ticket locks* and *MCS locks* provide fair FIFO ordering [24]. For *long* resource requests, we use mutexes as suspension-based mechanism instead of busy waiting. The mutexes use priority-ordering, following POSIX as example. This approach is also used in operating system kernels such as Linux. We further assume that synchronization mechanisms are used correctly. Preventing deadlocks should be done in user space and is outside of the scope of this paper.

Note that the presented approach is applicable to both single and multi-processor systems. A single processor system does not require the use of spinlocks. Additionally, the approach can be extended to partitioned EDF by using the *deadline floor protocol (DFP)* instead of IPCP [11, 2]. We plan to investigate this combination in future work.

To analyze the worst case, we do not perform an actual WCET analysis, as this requires detailed knowledge of the underlying processor architecture and the overall system, see e.g. [6]. Instead, we keep the worst-case considerations on an abstract level and identify the worst case for each building block in $\mathcal{O}$ notation. We think this is the right level to decide for or against a mechanism in general.

3 Previous Work

3.1 Efficient Synchronization Mechanisms and Futexes

The observation that system call overheads are expensive is well known. Several approaches were proposed to improve efficiency of synchronization mechanisms: Keedy describes atomic *test-and-increment* and *decrement-and-test* operations to implement uncontended semaphore operations and call into the operating system only to suspend or to wake up threads [18]. Birrell et al. describe an optimization for mutexes, condition variables, and semaphores in the Taos operating system to only call the operating system kernel when there is contention or a thread is waiting on a condition variable [4]. A similar approach is also described for *Benaphores* in BeOS [31]. For synchronization in a Java virtual machine, Bacon et al. proposed *Thin Locks*, based on atomic operations for uncontended cases, with a fall-back to OS provided synchronization primitives [3].

Futexes extend these prior approaches as a generic *compare-and-block* mechanism. Futexes allow a thread to *wait* on a variable in user space or to *wake up* a given number of waiting threads. The kernel dynamically creates an internal wait queue based on the given user space address and keeps the wait queue as long as threads are waiting. The third conceptual futex operation allows to *requeue* waiting threads from one wait queue to another. This helps when

signaling condition variables to prevent *thundering herd effects* [16]. The requeue operation transfers waiting threads from the condition variable's wait queue to the mutex' one instead of waking the threads up and letting them compete on the associated mutex.

Futexes were first introduced in Linux to implement POSIX thread synchronization objects in user space [14], and then later extended to support the priority inheritance protocol (PIP) [16]. Over time, scalability issues were addressed and discussed [8, 9]. Pizlo describes an approach resembling futexes using cascaded locks and hashed wait queues in user space for fine-grained locking and condition variables in the WebKit browser [28]. Spliet et al. evaluated the use of different real-time locking protocols for futexes in the context of LITMUS^{RT}, a Linux-based testbed for real-time scheduling experiments [34]. They use an index-based wait queue look-up and a bitmap of acquired locks that is shared between user space and kernel. Zuepke et al. presented approaches for deterministic futexes with FIFO ordering based on doubly-linked lists and look-up by thread ID [36], by replacing the futex hash table with binary search trees to bound interference effects to logarithmic runtime [38], and by using an index-based wait queue look-up [37].

3.2 Lock Holder Preemption

A problem with spinning synchronization in user space is that a thread can be preempted inside a critical section, as the scheduler is not aware of the critical section, and other threads continue spinning while the lock holder is preempted. In turn, multiple *preemption-safe lock mechanisms* were proposed [25], which either *prevent* preemption or try to *recover* from the fact that a lock holder is preempted, e.g. *adaptive mutexes*, by spinning only for a limited time and then falling back to blocking [26], scheduler hints [5], or *scheduler-conscious synchronization* with liveness indicators [20]. Michael and Scott provide an overview on these techniques [25]. We focus on mechanisms to prevent preemption.

Edler et al.'s *temporary non-preemption mechanism* in *Symunix II* uses two flags shared between user space and kernel [13]. Before starting spinning, a thread indicates its wish to disable preemption in the first flag. When the kernel actually wants to preempt the thread inside the critical section, the kernel sets the second flag to indicate a pending preemption request and let the thread continue. After the thread finishes the critical section, it clears the first flag and checks the second flag if it has to yield the processor. The kernel can set up a short timeout to enforce preemption of uncooperative threads not enabling preemption. An alternative approach is the *two-minute warning* mechanism proposed by Marsh et al. in *Psyche* [22]: the kernel indicates upcoming preemption (i.e. end of time slice) in a user readable flag, and a user space thread then avoids acquiring any spinlocks and rather yields. Holman and Anderson present a third approach in the context of *Pfair-scheduling* [17]: a locking attempt in the *frozen interval* at the end of a time slice implicitly blocks the thread until the next time slice. All three mechanisms were originally designed for systems with quantum scheduling where the time of preemption is known in advance.

3.3 Efficient IPCP Implementations

In the context of real-time systems with P-FP scheduling, lock holder preemption *by lower priority threads* can be also prevented by using real-time locking protocols such as IPCP. To reduce the overhead of frequent scheduling priority changes, Zuepke et al. presented two protocols to change a thread's priority lazily in user space [37]. The protocols use two variables shared between user space and kernel, similar to Edler et al.'s *temporary non-preemption mechanism* [13]. Almatary et al. presented an efficient implementation of IPCP (and DFP as well) using a different protocol with three shared variables [2].

Listing 1 User space per-thread data and UTCB

```
typedef struct {
    tid_t    tid;       // thread ID
    prio_t   uprio;     // user space priority
    prio_t   nprio;     // next thread's priority
    uint32_t ustate;    // user space state variable
    <...>
} thread_t;

#define SELF <...>      // get current thread's per-thread data
```

Listing 2 Priority change operations in user space

```
prio_t prio_raise(prio_t new_prio) {
    prio_t old_prio = SELF->uprio;
    assert(new_prio >= old_prio);
    SELF->uprio = new_prio;
    return old_prio;
}

void prio_restore(prio_t old_prio) {
    SELF->uprio = old_prio;
    if (old_prio < SELF->nprio) {
        sys_preempt();
    }
}
```

Listing 3 Priority change operations in the kernel

```
#define CURRENT <...>   // get current thread's kernel state (TCB)
#define UTCB <...>      // get current thread's user space data

void sys_preempt(void) {
    prio_t uprio = min(UTCB->uprio, CURRENT->max_prio);
    kernel_preempt(CURRENT, uprio);
}

void kernel_wake(<...>) {
    <...>
    prio_t uprio = min(UTCB->uprio, CURRENT->max_prio);
    prio_t nprio = ready_queue_next()->prio;
    UTCB->nprio = nprio;
    if (uprio < nprio) {
        kernel_preempt(CURRENT, uprio);
    }
    <...>
}

void kernel_preempt(thread_t *thr, prio_t prio) {
    // preempt the current thread
    <...>
}
```

All three protocols comprise a priority raise operation and a priority restore operation. The raise operations indicate a temporarily elevated scheduling priority in shared variables without using a system call. The restore operations revert the scheduling priority to the previous value and contain one optional system call to preempt the thread. On scheduling events, e.g. when releasing a suspended thread, the kernel obtains the elevated scheduling priority from the shared variables to consider whether to defer the preemption of the current thread.

In Zuepke et al.'s first protocol [37] and Almatary et al.'s protocol [2], the kernel also updates an in-kernel representation of the current thread's scheduling priority and then indicates that it has *observed* the priority change in the shared variables. In both protocols, the priority restore operations need a system call to update the in-kernel value again, even if the current thread will not be preempted.

Zuepke et al.'s second protocol addresses this shortcoming [37]. Here, the kernel does not update an in-kernel priority and indicates whether it has observed the priority change, but instead provides the priority of the *next* thread eligible for scheduling in a shared variable and updates its value on each scheduling event. With this, the restore operation only issues a system call when the current thread really needs to be preempted. We therefore focus only on this protocol. The two shared variables are named `uprio` (*user priority*) and `nprio` (*next thread's priority*).

Listing 1 shows the shared protocol variables `uprio` and `nprio` among other variables. We keep these variables in the per-thread TLS. Listing 2 shows the priority change operations in user space. The priority raise operation returns the previous scheduling priority for the later restore operation. Listing 3 shows the kernel parts of the implementation for the preemption system call and when waking up a suspended thread. In both cases, the user space priority is bounded to `max_prio` before further use.

Note that the kernel is optimized for frequent priority changes and does not keep the current thread on ready queues, so `nprio` is naturally available from the highest priority thread on the ready queue and also used internally by the kernel to decide whether to preempt the current thread or not. Blackham et al. describe a similar technique for seL4 [6].

From the point of view of their worse-case timing, all three protocols show equal behavior: no system call is needed to raise the priority, but a restore operation might require a system call[1], so we assume the system call is always called as the worst case. Also, all protocols show equal overhead to synchronize and validate user priorities when testing to preempt the currently running thread.

4 Light-Weight Blocking for IPCP

With the IPCP implementation presented in Section 3.3, we can now realize non-preemptive critical sections in user space. Listing 4 shows a simple example. A thread first raises its user space scheduling priority to `max_prio` to become non-preemptive, then acquires a spinlock. The threads previous priority is kept in `old_prio`. At the end of the critical section, the thread unlocks the spinlock and restores its previous scheduling priority. Note that this sequence does not need any system calls in the fast path. The system call to preempt the thread in `prio_restore` is only needed when in the meantime another thread with a medium priority higher than `old_prio` became ready.

[1] In Zuepke et al.'s protocols [37], the preemption system call would be superfluous if the thread is preempted after updating `uprio` but before calling `sys_preempt`. Almatary et al. solve this corner case at the expense of additional protocol variables [2]. In the worst case, one syscall is always needed.

Listing 4 Example non-preemptive critical section in user space

```
spin_t example_lock;

void example_cs(<...>) {
    prio_t old_prio = prio_raise(max_prio);
    spin_lock(&example_lock);
    <...>
    spin_unlock(&example_lock);
    prio_restore(old_prio);
}
```

Listing 5 System call implementation for light-weight waiting in IPCP

```
err_t sys_wait_at_prio(uint32_t *ustate, uint32_t cmp,
                       timeout_t timeout, prio_t wait_prio) {
    <...>
    spin_lock(&ready_queue_lock);
    uint32_t val = safe_user_space_read_access(ustate);
    if (val == cmp) {
        CURRENT->prio = min(wait_prio, CURRENT->max_prio);
        CURRENT->ustate = ustate;
        CURRENT->state = WAIT_USER;
        err = kernel_wait(CURRENT, timeout);
    } else {
        err = EAGAIN;
    }
    spin_unlock(&ready_queue_lock);
    <...>
}
```

We now extend the critical sections with a blocking mechanism that interacts properly with the IPCP implementation and the spinlock-protected critical sections. We opt to manage the wait queue of blocked threads in user space.

Waiting: We can make the following considerations for a waiting operation:

- Suspending the current thread needs help by the kernel. This requires a system call.
- Waiting *after* calling `prio_restore` could trigger unnecessary preemption, as the calling thread is going to suspend itself anyway. Waiting at `max_prio` is advisable.
- Waiting *inside* the spinlock-protected critical section causes problems, as other threads would be unable to acquire the spinlock. A system call to suspend a thread must happen *after* unlocking the spinlock in user space.
- As the spinlock-protected critical section protects any internal state w.r.t. blocking, a system call to suspend a thread *outside* the critical section must prevent *missed wake-ups*.
- A thread's scheduling priority at wakeup time should reflect its original priority. When a thread is woken up at `max_prio`, it would execute only to the point where it restores its original priority and then causes unnecessary context switches if other medium priority threads are ready.
- Spurious wake-ups, e.g. timeouts, require a second critical section *after* waiting to remove the current thread from the wait queue again.

Listing 6 System call implementation for light-weight wake-up of a thread in IPCP

```
err_t sys_wake_set_prio(tid_t tid, uint32_t *ustate, uint32_t cmp,
                        prio_t new_prio) {
    <...>
    thread_t *thr = kernel_lookup_TCB_by_tid(tid);
    spin_lock(&ready_queue_lock);
    new_prio = min(new_prio, CURRENT->max_prio);
    UTCB->uprio = new_prio;
    uint32_t val = safe_user_space_read_access(ustate);
    if ((thr != NULL) && (thr->state == WAIT_USER)
                      && (thr->ustate == ustate) && (val == cmp)) {
        kernel_wake(thr);
    } else if (new_prio < UTCB->nprio) {
        kernel_preempt(CURRENT, new_prio);
    }
    spin_unlock(&ready_queue_lock);
    <...>
}
```

To prevent missed wake-ups, we use a *compare-and-block* mechanism similar to futexes. Inside the spinlock-protected critical section, a thread decides to block and reads a state variable. The state variable encodes a waiting condition and is changed by a wakeup operation. Then the thread unlocks the critical section in user space and calls the kernel to suspend. The kernel reads the state variable again and, if the current value matches the previous value, suspends the thread. An alternative would be to call the kernel from inside the critical section and let the kernel unlock the critical section before suspending the calling thread. This would prevent any ambiguity with parallel wakeup operations. However, the kernel would then need to know the exact semantics of the spinlocks to unlock the spinlock for the caller. We opt to unlock the spinlock in user space instead. This keeps the kernel simple.

To address the problems of the scheduling priority at wakeup time, we temporarily *drop* the priority while waiting. While still executing at `max_prio` in user space, a thread calls a system call to wait at a lower priority `wait_prio`. The kernel then temporarily sets the thread's priority to `wait_prio` while waiting. When the thread is woken up again, it will be enqueued at `wait_prio` on the ready queue. And when the thread is eventually scheduled, the kernel increases the scheduling priority back to `max_prio`, and then returns from the waiting system call. We can easily achieve this by using the following trick in the IPCP implementation of Section 3.3: the kernel lets the thread wait at `wait_prio`, but leaves `uprio` unmodified while waiting. Note that `uprio` was set to `max_prio` before waiting, so the thread is effectively running at `max_prio` again after waiting as well. The thread in user space can then either lock the spinlock again, or leave the IPCP-protected critical section and restore its previous scheduling priority `old_prio`. As no other threads with a higher priority will be ready at that moment, no system call will be needed.

Listing 5 shows the implementation of the `sys_wait_at_prio` system call. With the ready queue locked, the kernel evaluates if the content of a given state variable in user space (`ustate`) matches a compare value (`cmp`). If this is the case (no missed wakeup), the kernel keeps the address of `ustate` for later, and suspends the current thread with the given timeout (`timeout`) on the given waiting priority (`wait_prio`) in an internal waiting state `WAIT_USER`. In case of a missed wakeup, the kernel returns an error condition. The system call does not change `uprio`, so the thread will be immediately boosted to its previous `uprio` after wakeup.

Wakeup: For a wake-up operation, we can discuss similar considerations as for waiting:

- Waking up a waiting thread needs help by the kernel. This requires a system call as well.
- The wakeup system call addresses a blocked thread directly by its thread ID (`tid`).
- Waking a thread up *after* calling `prio_restore` could cause unnecessary delays due to preemption. Again, doing the wake-op operation at `max_prio` is advisable.
- The wakeup system call could happen *inside* the spinlock-protected critical section or be deferred after unlocking the spinlock. In the latter case, a system call to wake up a thread *outside* the critical section must prevent *spurious wake-ups*.
- Waking up a thread with a priority higher than oneself causes preemption when restoring the previous priority.

We opt to wake up a thread outside the critical section. To prevent *spurious wake-ups*, we use same technique as in the waiting operation. User space code passes the address and expected value of a state variable in user space to the system call, and the kernel compares the state variable to the expected value and only unblocks the given thread on a match. This constitutes a *compare-and-unblock* mechanism.

To prevent unnecessary system calls for preemption in `prio_restore`, we defer the wake-up to the latest possible point and *fuse* the system call for wake-up with the system call for preemption. The resulting system call combines wake-up, priority change, and preemption.

Listing 6 shows the implementation of the `sys_wake_set_prio` system call. The wake-up operation directly references a waiting thread by its thread ID (`tid`), so the system call first looks up the TCB in the kernel. With the ready queue locked and executing non-preemptively, the kernel first bounds the given priority to restore and updates `uprio` in user space. Then the kernel evaluates the wake-up condition: the thread must exist, it must be waiting in a call to `sys_wake_set_prio`, it must wait on the same `ustate` variable, and the current value of `ustate` must match a compare value (`cmp`). If the condition is met, it wakes up the thread. Otherwise, it just preempts the current thread, if necessary. Recall that `kernel_wake` in Listing 3 also preempts the current thread when a higher priority thread is woken up.

5 Non-Preemptive Busy-Waiting Monitors in User Space

We now discuss how to construct higher-level synchronization mechanisms based on the IPCP implementation of Section 3.3 to temporarily disable preemption, fair spinlocks, and the light-weight waiting and wake-up primitives of Section 4. We use the term *monitor* for the resulting design because the operations resemble the ones found in monitor implementations.

A monitor protects the *state* of a specific synchronization object of a higher-level synchronization mechanism, e.g. the current lock owner of a mutex, and one or more wait queue heads. We place the according wait queue nodes in each thread's TLS segment, or on the stack in the waiting functions. As a thread can only wait on one wait queue at a time, the space for wait queue nodes is bounded.

A thread *enters* the monitor to gain *exclusive access* to the internal state, and *leaves* the monitor afterwards. When multiple threads try to enter the monitor, they are serialized by the spinlock. This relates to a FIFO-ordered *enter queue*. Within the monitor, a thread can decide to *wait* or to *notify* waiting threads. Waiting effectively comprises enqueuing the thread in a wait queue, leaving the monitor, blocking in the kernel, and entering the monitor again after waiting. To handle spurious wake-ups, a thread must eventually remove itself from the wait queue. For notification, a thread removes a blocked thread from the wait queue and wakes up up the blocked thread when leaving the monitor. The design is

optimized to perform an uncontended *enter* → *leave* sequence without system calls, and both *wait* and *notify* with one system call in the best case. Note that more than one thread can be woken up, but this requires one additional system call for each thread.

Another important aspect is to handle the `ustate` variables correctly. For this, we draw from *eventcounts* and *sequencers* [30]. Firstly, we will use `ustate` as a counter that is incremented *before* a thread suspends or is woken up. The motivation to use a counter instead of a binary state like `WAITING` and `READY` is to prevent spurious wake-ups due to *ABA-problems* when two waiting operations follow each other back to back. Secondly, we will use a dedicated `ustate` variable for each thread. Like a *sequencer*, the increments of a thread's `ustate` variable in user space order the particular wait and wakeup operations of the related thread. The waiting operation in the kernel follows *eventcounts*. As we use a dedicated per-thread counter, a wait operations will observe at most one additional increment from the corresponding wakeup operation. With this, the *compare-equal* condition for blocking in the kernel is sufficient to detect missed wake-ups. The increment before waking up a thread also follows *eventcounts*. As any further waiting attempt would increment `ustate` again, the *compare-equal* condition for unblocking in the kernel is sufficient to prevent spurious wake-ups. Note that we keep `ustate` in the TLS segment of each thread, see Listing 1.

The `ustate` variables are only modified in the critical sections of the related synchronization objects in user space. Nevertheless, an implementation should change `ustate` by using atomic read and write operations to prevent undefined behavior by the compiler, as the kernel reads the current value in parallel.

6 Analysis of Designs for Blocking Synchronization Mechanisms

6.1 Building Blocks

To compare the presented monitor approach to futexes and a traditional implementation using dedicated system calls, we first analyze how blocking synchronization mechanisms are typically implemented in operating systems using fine-grained locking like Linux. For this, we define a *generic blocking mechanism* and decompose it into its internal building blocks. The generic blocking mechanism provides two operations. The *waiting* operation either lets the calling thread continue or suspends it, and the *wake-up* operation can optionally wake up a previously suspended thread. This resembles common operations on mutexes, semaphores, or condition variables, but without specifying the *exact* semantic of the synchronization mechanism. We denote a step where the exact semantics of an actual synchronization mechanism would be required as *semantic operation*. Note that futexes and the monitor approach require two semantic operations, a 1st semantic operation in user space and a 2nd semantic operation in the kernel. The system call approach needs one only in the kernel.

Table 1 shows a comparison of the generic blocking synchronization mechanism implemented as (i) a system call based approach as baseline, (ii) futexes like in Linux, and (iii) the proposed monitors. The upper part of the table shows the layered individual steps to suspend or to wake up a thread from top to bottom. A "•" marks the operations when no blocking is needed, i.e. the fast path. The lower part of the table shows associated data in user space and the kernel. Here, a "◇" indicates global data. Comparable operations and data objects are placed in the same rows.

System call: In the baseline implementation using system calls, user space code calls the kernel with an identifier to a kernel object. In turn, the kernel validates the identifier and retrieves the kernel object comprising all necessary data in the look-up step. Then the kernel

Table 1 Comparison of three implementations of a generic blocking synchronization mechanism. The upper part shows the layered operations from user space down to the kernel. A " • " marks the operations in the fast path when no blocking is needed. The lower part shows the associated data in both user space and the kernel. A "◇" denotes global data.

operations	**system call (baseline)**	**futex**	**monitor (this paper)**
user space			• disable preemption
			• lock user space object
		• 1st semantic op. (atomic)	• 1st semantic operation
			wait queue operation
	• system call	system call	system call
kernel	• look-up kernel object	look-up futex wait queue	look-up thread
	• disable preemption	disable preemption	disable preemption
	• lock kernel object	lock wait queue	
	• semantic operation	2nd semantic operation	
	wait queue operation	wait queue operation	
	lock ready queue	lock ready queue	lock ready queue
			2nd semantic operation
	suspend / wake-up	suspend / wake-up	suspend / wake-up

data model	**system call (baseline)**	**futex**	**monitor (this paper)**
user space	ID of kernel object	futex value (atomic)	user space object lock
		additional semantic state	semantic state
			wait queue
			◇ thread states
kernel	kernel object lock	wait queue lock	
	semantic state	address	
	wait queue	wait queue	
	◇ ready queue lock	◇ ready queue lock	◇ ready queue lock
	◇ thread states	◇ thread states	◇ thread states

disables preemption, locks the data in the kernel object, and performs a semantic operation, such as checking and modifying the internal state. At this point, the semantic operation decides whether the operation is completed or if a wait queue operation is needed. In the latter case, the kernel either enqueues the calling thread on the wait queue, or removes a waiting thread from the wait queue, depending on the desired operation. Then the kernel must lock internal scheduling data (ready queue lock) before it can finally suspend the calling thread or wake up a waiting thread.

Futex: Compared to the system call approach, the main difference of the futex-based implementation is the 32-bit variable in user space expressing the semantic state. Depending on the atomic operation on the variable, the 1st semantic operation either succeeds immediately or requires a system call. In the kernel, the look-up of an associated in-kernel wait queue is based on the user space address of the atomic variable, but the following steps are similar to the baseline approach. That is because futexes *are* a generic compare-and-block mechanism. The 2nd semantic operation in the kernel checks if the futex value has changed in the meantime. This indicates a parallel wake-up operation, and the system call returns in this case, similarly to the baseline version.

Monitor: The monitor-based implementation differs from both approaches. The user space object already comprises a lock, semantic state, and a wait queue. User space code disables preemption and locks the object before it evaluates the internal semantic state in the 1st semantic operation. In the fast path, the 1st semantic operation succeeds and the operation completes. In the slow path, a system call is required to block or to wake up. For blocking or wake-up, the wait queue operation either adds the current thread to the wait queue or removes a thread from the wait queue and then in turn calls into the kernel. The kernel first validates the given thread ID and locates the target thread. Then the kernel locks the necessary scheduling data and performs a 2nd semantic operation. The 2nd semantic operation is serialized by this last lock and detects parallel wake-up or suspend operations. If its semantic check succeeds, a suspend or wake-up operation takes place.

6.2 Analysis of the Fast Paths

Note that the overhead in the fast path in user space for both the futex and the monitor variants is less than a system call, but the monitor shows more overhead than a futex. A futex fast path typically comprises one atomic operation with either acquire or release semantics or equivalent memory barriers. The monitor fast path requires to disable preemption (load and store instructions on the local processor), and one or two atomic operations with both acquire and release semantics or equivalent memory barriers in the spinlock operations.

When comparing the three implementations shown in Table 1, we can see that both the futex and the monitor require additional operations compared to the baseline approach. If we consider the atomic operation in the futex case as a (somewhat "compressed") critical section, then both futex and monitor use a critical section in user space to guard the fast path with the 1st semantic operation. However, there is a second critical section in the kernel as well to guard the 2nd semantic operation that decides whether to block or unblock.

The key technique for the separation into a fast-path in user space and the actual blocking/unblocking in the kernel is to use two serialized critical sections, one in user space and one in the kernel. These critical sections are *loosely coupled* by semantic state data that is *set* in the 1st semantic operation and *checked* in the 2nd one. The benefit of this pattern is that one can determine the WCET of each critical section in isolation.

The 2nd semantic operation in the kernel prevents race conditions between suspend and wake-up operations ongoing in parallel. In the common case, the 2nd semantic operation simply succeeds, but how do the implementations behave if the 2nd semantic operation fails? Non-real-time futexes in Linux use *compare-equal* semantics. If the futex value in user space no longer matches a given previous value, the kernel does not suspend the calling thread, but returns to user space. The user space part then *retries* the whole operation from the beginning. For priority inheritance (PI) mutexes, the Linux kernel solves this situation internally and tries to lock a mutex for the calling thread *again.* In both cases, the loops for this are potentially unbounded. The monitor approach naturally works without any retrying.

6.3 Analysis of the Worst Case

From a worst-case point of view, where we assume that the fast paths are not taken, we see mostly similar operations for all three variants in Table 1. For a typical implementation, we can further assume that a system call, a look-up of a pre-registered kernel object, suspending the current thread, or a wake-up of a blocked thread need $\mathcal{O}(1)$ time. Locking operations usually have $\mathcal{O}(m)$ worst-case behavior for m processor cores. Wait queue operations take at most $\mathcal{O}(\log n)$ time when using balanced binary search trees or $\mathcal{O}(n)$ when using sorted

linked lists for n blocked threads. For the semantic operations, we can also assume $\mathcal{O}(1)$ timing behaviour, e.g. checking the state of a mutex and either making the thread the new mutex owner or suspending the thread.

When we compare the futex approach to the baseline, we see that using futexes adds an atomic operation in user space and requires a more complex look-up operation in the kernel. Hashed wait queues are fast, but have determinism issues and degrade to $\mathcal{O}(n)$ in the worst case. In [38], the author describes a deterministic approach by using a balanced binary search trees with $\mathcal{O}(\log n)$ time for look-up of wait queues. A critical point is to prevent loops if the second semantic operation fails, as this is the case for futex operations on real-time-mutexes using the priority inheritance protocol in Linux. These potentially unbounded loops make futexes hard to assess in a WCET analysis.

The monitor approach moves some steps of the baseline version from the kernel to user space and adds just one additional semantic check before suspending or waking up a thread. The critical point is the non-preemptive critical section in user space. An in-kernel implementation can simply disable interrupts to achieve non-preemptiveness, however, this is not possible in user space. Therefore, in a WCET analysis, extra delays due to interrupt handling have to be accounted for, e.g. see [7] for applicable techniques. Additional overhead is caused by the priority restore operation, where we must account an additional system call to preempt the thread after the critical section in user space. We can model this overhead as a constant. All in all, a WCET analysis of the kernel parts of the monitor approach should be simpler than for the baseline approach, but the user space parts require to account for additional overheads of interrupt handling and preemption.

7 Implementation of Blocking Mutexes and Condition Variables

7.1 Blocking Mutexes

Based on the monitor building blocks of Section 5, we now present a blocking mutex. The data structure representing a mutex comprises a spinlock, an owner field, and a wait queue.

Listing 7 shows a mutex lock operation with timeout in `mutex_lock`. The function first raises the scheduling priority to `max_prio` and locks the internal spinlock. If the mutex is currently unlocked, the function registers the calling thread as mutex owner and returns successfully after unlocking the spinlock and restoring the previous priority. Otherwise, the function adds the current thread to the priority-ordered wait queue using its original priority. Then the mutex function retrieves and increments the current value of its user state variable `ustate`, unlocks the spinlock, and suspends itself in the kernel with the timeout and its original priority. After wake-up, the function locks the spinlock again and tests if `ustate` was incremented. If true, the current thread is now the lock owner. If not, the timeout has expired instead, and the function removes the thread from the wait queue. The function unlocks the spinlock, restores the previous priority and returns the status of lock ownership.

The unblock operation is shown in `mutex_unlock` in Listing 7. Again, the function increases the scheduling priority and locks the spinlock. Then it tries to retrieve the highest priority waiting thread from the wait queue. If no thread is found, the function sets the mutex to unlocked state, unlocks the spinlock, restores the previous priority, and returns. Otherwise, the operation makes the waiting thread the new lock owner, increments its user mode state variable, unlocks the spinlock, and performs a fused wake-up and priority restore operation.

In the best case, when we assume that the priority restore operation does not preempt the thread, then both mutex lock and unlock operations do not need any system calls in the fast path, and only one system call for blocking and wake-up on contention. This is similar

Listing 7 Implementation of a blocking mutex with timeout and a priority-ordered wait queue

```
typedef struct {
    spin_t lock;        // internal spinlock
    tid_t owner;        // current mutex owner or UNLOCKED
    waitq_t waitq;      // priority-ordered mutex wait queue
} mutex_t;

bool_t mutex_lock(mutex_t *m, timeout_t timeout) {
    prio_t old_prio = prio_raise(max_prio);
    spin_lock(&m->lock);

    bool_t success = (m->owner == UNLOCKED);
    if (success == TRUE) {
        m->owner = SELF->tid;
        goto out;
    }

    waitq_add_ordered(&m->waitq, SELF, old_prio);
    uint32_t seq = ++SELF->ustate;
    spin_unlock(&m->lock);
    sys_wait_at_prio(&SELF->ustate, seq, timeout, old_prio);
    spin_lock(&m->lock);

    success = (SELF->ustate != seq);
    if (success == FALSE) {
        waitq_remove(&m->waitq, SELF);
    }

out:
    spin_unlock(&m->lock);
    prio_restore(old_prio);
    return success;
}

void mutex_unlock(mutex_t *m) {
    assert(m->owner == SELF->tid);
    prio_t old_prio = prio_raise(max_prio);
    spin_lock(&m->lock);

    thread_t *next = waitq_remove_highest(&m->waitq);
    if (next == NULL) {
        m->owner = UNLOCKED;
        spin_unlock(&m->lock);
        prio_restore(old_prio);
        return;
    }

    m->owner = next->tid;
    uint32_t seq = ++next->ustate;
    spin_unlock(&m->lock);
    sys_wake_set_prio(next->tid, &next->ustate, seq, old_prio);
}
```

to futexes. However, in the worst case, we must assume that the mutex is contended and the thread is interrupted, and then we need to account two system calls for `mutex_lock` (one for waiting, one for preemption) and one system call for `mutex_unlock` (combined wake-up and preemption). Compared to futexes and the baseline, we have to account one additional system call for preemption in `mutex_lock`.

7.2 Condition Variables for Blocking Mutexes

We present condition variables with POSIX semantics for the mutexes described in Section 7.1. Both `cond_wait` and `cond_notify` operations expect a locked support mutex. The data type of the condition variable comprises just a wait queue. The internal spinlock of the support mutex also protects the wait queues of condition variables. Discussing the implementation in detail exceeds the page limitation of this paper, so we provide only a brief overview.

`cond_wait` enqueues the calling thread on the wait queue of the condition variable, unlocks the mutex, then waits. `cond_notify` simply requeues the given number of threads (one or all) from the wait queue of the condition variable to the wait queue of the mutex. After wake-up, a thread is either the mutex owner (successfully notified and requeued to the mutex), or not (spurious wake-up, e.g. timeout). In the latter case, the thread blocks again on the mutex.

In the best case (no preemption when restoring the previous priority), waiting needs one system call and notifying needs no system call at all, as threads just get requeued to the mutex wait queue and the actual wake-up is done when the notifying thread unlocks the mutex. Therefore, we must include the mutex operations in the discussion as well. Now a *lock* → *wait* → *unlock* sequence takes at most one system call to wait for the condition, and a *lock* → *notify* → *unlock* sequence also needs only one when unlocking, regardless of the number of notified threads. Again, this is similar to futexes. In the worst case (with preemption), we must account *five* system calls for waiting: one to lock the mutex, one when unlocking the mutex and waking up the next mutex owner, one for waiting on the condition variable, another one to block on the mutex again on spurious wake-ups, and the last one for preemption when unlocking the mutex. Futexes require four system calls (the final one for the preemption is not needed). For notification, the number of required system calls is two, one to lock the mutex and one to notify. This is the same for futexes. In both best and worst case, the baseline version always needs three system calls for these sequences. Also, moving threads between queues has the same overhead in all three versions.

8 Experimental Evaluation

For comparison, we have implemented all three approaches (baseline, futex, and monitor) in a small real-time operating system (RTOS) named *Marron*. Marron provides static partitioning of OS resources with fixed-priority scheduling on each processor core. The kernel implements fine-grained locking with a similar implementation complexity as described in Table 1.

Marron currently supports only 32-bit ARM platforms, so we evaluated the approaches on three system-on-a-chip platforms. A *BeagleBone Black* provides a single Cortex A8 core running at 550 MHz, a *Freescale i.MX6Q* has four Cortex A9 cores at 792 MHz each, and the *BeagleBoard-X15* has two Cortex A15 cores at 1 GHz. Note that the Cortex A8, A9, and A15 cores have different microarchitectures. The A8 is an in-order design with a 13-stage pipeline, while the A9 and A15 are out-of-order designs, with a short 8-stage pipeline on the A9 and a longer 15-stage pipeline on the A15. As the working set is small, we expect our experiments to fit into both instruction and data caches and not access any external DRAM. Also, we run our experiments without any interference from other applications. For better

Table 2 Overhead measurements on 32-bit ARM platforms in CPU cycles. The first set of measurements determines the overhead of the building blocks, the last four sets compare the different approaches for synchronization mechanisms in uncontended and contended scenarios.

Test		Cortex A8	Cortex A9	Cortex A15
acquire and release barriers		33	6	20
CAS without barriers		8	22	30
uncontended spin_lock/unlock pair		74	38	108
prio_raise/restore pair		17	12	6
null system call		177	107	205
uncontended mutex lock/unlock pair	system call	711	465	742
	futex	110	87	175
	monitor	218	147	260
contended mutex lock/unlock pair	system call	1557	1137	1503
(same core)	futex	2051	1838	1995
	monitor	1792	1668	1689
contended mutex trylock/unlock pair	system call	—	1172	1696
(2 cores)	futex	—	302	495
	monitor	—	712	1087
contended mutex trylock/unlock pair	system call	—	3182	—
(4 cores)	futex	—	779	—
	monitor	—	2445	—

comparison, we present the results of each CPU core in clock cycles. Measurements were taken with the internal cycle counter of the CPU cores. We ran each measurement in a loop 1024 times and divided the result, therefore all measurements include the loop overhead.

We performed an experiment to evaluate the fast-path performance of the three discussed approaches and their building blocks in different contention scenarios Table 2 shows the results for the three different processor cores. The overhead of the building blocks in isolation and the uncontended mutex runs show stable results, as these tests run in a single-threaded context. We determine the results for the contended case on the same processor core indirectly with the help of a second thread. The presented results show the remaining overhead of the contended mutex operations and two additional context switches. The contended case on different processor cores determines the overhead of the operations in user space, e.g. the internal critical sections of the monitor, on two or four processor cores in parallel. The test uses `mutex_trylock` instead of a blocking system call and effectively spins until it successfully acquires the mutex. The test releases two resp. four cores from a barrier and measures the time until all cores acquire and release the mutex once, and reach the barrier again. The baseline variant implements `mutex_trylock` as system call, futexes use atomic operations in user space, and the monitor approach uses an internal critical section. These results show a great variation between runs and should be treated as a rough indicator of what to expect.

Observation 1: The futex approach is faster than the monitor approach, and both are faster than the baseline approach using system calls. The system call overhead dominates everything else. We expected to see a similar ratio of the performance of atomic operations to system calls on ARM as reported by [1] and [33] for x86 processors, and our experiments show this. Note that the measurement of the a system call includes the overhead of the operating system to save and restore registers on kernel entry and exit. Therefore, our approach to avoid system calls is reasonable.

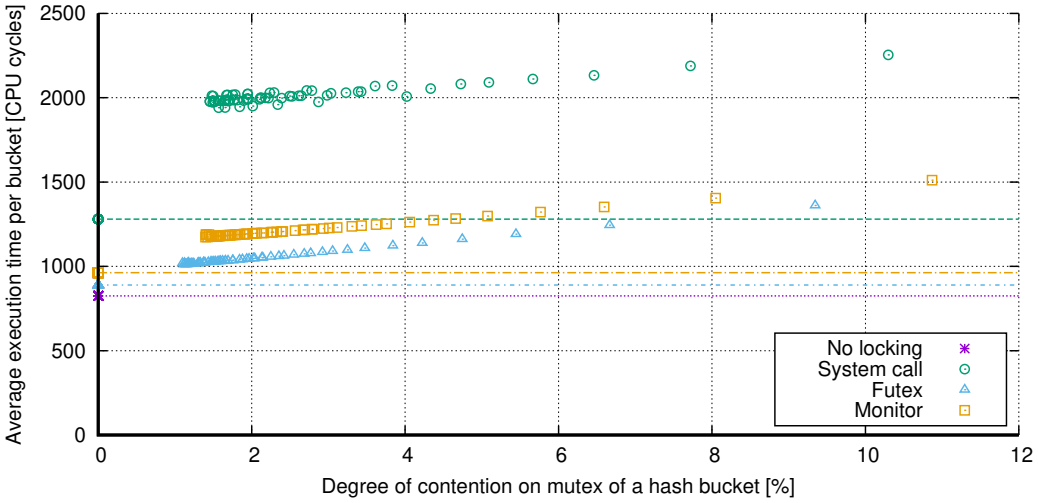

Figure 1 Operations of varying degrees of contention on a shared hash table with 64 hash buckets, each protected by a mutex. Each thread keeps a hash bucket locked for a constant time of 1 µs.

Observation 2: Efficient implementations of IPCP do not contribute much overhead to the fast path. On all three platforms, changing priorities in user space does not cause much overhead. The measurement of the priority raise/restore pair shows this.

Observation 3: On ARM processors, atomic operations cause significant overhead. The measurement of a pair of lock and unlock operations on a ticket spinlock shows significant overhead on the Cortex A15. We did not expect this. As ARM processors use a weakly ordered memory model and require explicit memory barriers to order memory accesses, we investigated this further and measured memory barriers and CAS operations in isolation (first three rows). When using futexes, a mutex lock / unlock pair comprises a sequence of CAS + acquire barrier + release barrier + CAS, and the parts add up correspondingly. Similarly, the monitor requires two pairs of spinlock lock / unlock operations of equal complexity than a futex pair, explaining the more than twice as high overhead of the monitor in the fast path. Note that the baseline version shows a similar locking overhead inside the kernel.

Observation 4: On contention on the internal critical section of the monitor, the monitor shows worse results than futexes. Here, threads in `mutex_trylock` repeatedly spin to lock the internal critical section of the monitor to detect that the mutex is already taken. Futexes fare well here, as they can directly probe the mutex due to the atomic operations. And implementing `mutex_trylock` by repeated system calls is not a reasonable design choice.

Observation 5: In the contended case with blocking, the system call approach shows the least overhead. Futexes and monitors show more overhead, as they first detect contention in user space before calling into the kernel. Monitors are slightly faster than futexes due to the simpler kernel implementation. This stresses the point that the fast-paths in the uncontended case come with extra costs in the contended case.

We conduct another experiment to compare the three discussed approaches in a scenario of varying degrees of contention. For this, we distribute 2^{24} random values following a square distribution to a shared hash table comprising 64 hash buckets. We run this experiment using four parallel threads (one for each core) on the Cortex A9. Each thread atomically draws a

unique value from the random pool and locks the resulting hash bucket for a constant time of 1 µs. Statistics counters in the mutex implementations account contended and uncontended cases. Figure 1 shows the average execution time per bucket operation in CPU cycles (incl. locking overhead) for the varying degrees of contention observed in the hash buckets. We include the results of a run without any locking and without contention as reference shown as dotted horizontal lines. Note that 1 µs relates to 792 cycles on the Freescale i.MX6Q platform. The results show that both futexes and monitors result in less overhead in low contention scenarios compared to the system call approach. Also, futexes show less overhead compared to the monitor. A second effect is that both futexes and monitors show *less* contention than the system call approach. Recall that both approaches comprise *two* semantic checks whether to suspend the current thread. The second semantic check in the kernel provides a *second chance* to acquire the mutex after a brief delay (the system call overhead). Again, this effect is stronger in futexes.

9 Discussion

In general, the evaluation shows that the monitor approach works and saves CPU cycles by avoiding system calls in the uncontended case. The monitor has similar properties as futexes. Synchronization mechanisms built on top do not need initial registration in the kernel, and therefore also no resources or memory allocations in the kernel. The analysis in Section 6.3 shows that the monitors are better than futexes w.r.t. determinism, but they also come with more overhead due to the IPCP and spinlocks to protect internal critical sections, as the evaluation in Section 8 shows.

Note that we only did microbenchmarks to compare specific details in a hot-cache scenario, so the question is how big will the performance win be for a real application. This generally depends on the specific type of application. Both real-time and non-real-time applications originally designed for single-processor systems will probably not benefit much, as they are usually carefully tuned to avoid synchronization overhead in the first place. However, the situation is different if we consider multi-threaded applications with lots of fine-grained locking and low contention, e.g. language environments for Java [3] or JavaScript [28]. While these are *not* typical real-time applications, we can expect that such applications will be deployed in mixed criticality environments, so real-time operating systems should prepare to handle best-effort workloads efficiently as well.

From a WCET point of view, the monitor approach reduces determinism issues compared to futexes, as the analysis in Section 6.3 shows. Mutexes based on monitors do not require loops in the kernel or in user space, and the look-up of a particular thread is simpler than the look-up of a wait queue in futexes. As the monitor building blocks are similar to the baseline version but just shifted in place, the worst-case considerations are similar for both. The monitor adds additional constant overheads for the extra system calls for preemption. However, only the wake-up of one thread is optimized and interacts nicely with IPCP. Waking up an additional thread needs one additional system call each. But this only affects operations waking up multiple threads, e.g. when using a `barrier_wait` operation where the last thread arriving at the barrier wakes up all waiting threads. The system calls to wake-up the additional threads happen inside the critical section of the monitor and must be accounted to the WCET of the monitor as well.

Due to direct addressing of threads, the monitor approach is limited to synchronization of threads in the same process, if we assume that thread ID are local to a single process and not globally accessible. But this is the typical use case for thread synchronization in most applications anyway and therefore not a problem. However, if a system allows access to

threads in other processes, then the monitor approach can also be used for shared memory communication, like futexes. In this case, a thread's waiting state variable (`ustate`) must be placed in the shared memory as well. Note that the robustness considerations here are the same as when using futexes or other synchronization mechanisms, as synchronization over shared memory requires that applications must trust each other. But typically, access to threads in other processes is a source of unwanted interference and therefore not possible. We also expect that threads behave correctly and use the protocols appropriately, but the impact on other processes is bounded by `max_prio`. Still, further mechanisms to detect runaway threads are possible. For example, the kernel could set up a timer when it has to defer a preemption request, and the system call for preemption clears the timer again.

From a security point of view, the efficient implementation of IPCP can leak scheduling information of unrelated processes if processes are not temporally isolated, e.g. by a TDMA scheme like in ARINC 653 for avionics. In the used protocol, `nprio` exposes the priority of the next eligible thread for scheduling on the ready queue to other processes.

Lastly, the presented mutexes do not address priority inversion. The monitor approach can only implement synchronization mechanisms using *anticipatory* locking protocols [34], such as IPCP, where the locking protocol prevents problematic situations a priori. As the monitor approach already uses IPCP internally, a `mutex_lock` operation can easily implement IPCP mutexes by not restoring a thread's previous scheduling priority, but by setting the priority to the ceiling priority of the mutex after successfully locking the mutex. But also other priority-based mechanisms to address priority inversion should work, for example MPCP [29]. For MPCP, the priority space in user space needs to be partitioned into a range of normal priorities and a boosted priority range, with the non-preemptive priority on top. Similar to the IPCP example, a lock holder then uses the boosted ceiling priority of the mutex until release. Note that these approaches also work for the futexes in our setting, as they only depend on the efficient IPCP implementation. *Reactive* locking protocols [34], such as PIP, are not suitable for the monitor approach, as the kernel lacks the necessary information to build a resource allocation graph, and handling the priority changes for PIP in user space would require additional system calls. When using PIP, futexes are the perfect choice, as the protocol activates on resource contention, and this is the case where futexes need system calls anyway.

10 Conclusion

In this paper, we explored the design space of blocking synchronization mechanisms optimized to avoid costly system calls in the fast, uncontended path. We analyzed different design approaches using futexes and monitors and compared them to a system call based approach. For each mechanism, we discussed the impacts on determinism and related WCET concerns. We presented a light-weight monitor implementation in user space comprising efficient implementations of IPCP, spinlocks, and interwoven blocking mechanisms. The proposed monitor approach avoids the determinism issues of the futexes by avoiding unbounded loops and a complex look-up mechanism for wait queues in the operating system kernel. Furthermore, compared to the simple atomic variable state used by futexes, the monitor approach enables a richer semantics in critical sections. The experimental evaluation shows that both monitors and futexes considerably reduce the overhead compared to the baseline system call approach. The extra complexity of the monitor is reflected in the additional overhead of this approach compared to the futex one. The presented approach is suitable as building block to construct other blocking synchronization mechanisms in RTOS implementations for multi-processor real-time systems with P-FP scheduling.

References

1 Samy Al-Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, 2013. doi:10.1145/2483852.2483866.

2 Hesham Almatary, Neil C. Audsley, and Alan Burns. Reducing the implementation overheads of IPCP and DFP. In *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, pages 295–304. IEEE Computer Society, 2015. doi:10.1109/RTSS.2015.35.

3 David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM. doi:10.1145/277650.277734.

4 Andrew Birrell, John V. Guttag, James J. Horning, and Roy Levin. Synchronization primitives for a multiprocessor: A formal specification. In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 94–102. ACM, 1987. doi:10.1145/41457.37509.

5 David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990. doi:10.1109/2.53353.

6 Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pages 339–348. IEEE Computer Society, 2011. doi:10.1109/RTSS.2011.38.

7 Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

8 Neil Brown. In pursuit of faster futexes. *LWN*, May 2016. URL: https://lwn.net/Articles/685769/.

9 Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. *LinuxCon North America, Chicago, IL*, August 2014. URL: http://events17.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

10 Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, March 1995. doi:10.1145/214037.214100.

11 Alan Burns, Marina Gutiérrez, Mario Aldea Rivas, and Michael González Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Trans. Computers*, 64(5):1241–1253, 2015. doi:10.1109/TC.2014.2322619.

12 Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.

13 Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, pages 1–17, September 1988.

14 Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the 2002 Ottawa Linux Symposium*, pages 479–495, 2002.

15 Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, 2-6 December 2001*, pages 73–83. IEEE Computer Society, 2001. doi:10.1109/REAL.2001.990598.

16 Darren Hart and Dinakar Guniguntalay. Requeue-PI: Making Glibc Condvars PI-Aware. In *Eleventh Real-Time Linux Workshop*, pages 215–227, 2009.

17 Philip Holman and James H. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*, pages 149–158. IEEE Computer Society, 2002. doi:10.1109/REAL.2002.1181570.

18 James Leslie Keedy. An outline of the ICL 2900 series system architecture. *Australian Computer Journal*, 9(2):53–62, 1977.

19 Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. doi:10.1109/SP.2019.00002.

20 Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.*, 15(1):3–40, 1997. doi:10.1145/244764.244765.

21 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

22 Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 110–121, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/121132.344329.

23 Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019. arXiv:1902.05178.

24 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. doi:10.1145/103727.103729.

25 Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998. doi:10.1006/jpdc.1998.1446.

26 John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*, pages 22–30. IEEE Computer Society, 1982.

27 John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990*, pages 247–256. USENIX Association, 1990.

28 Filip Pizlo. Locking in webkit. online article, May 2016. URL: https://webkit.org/blog/6161/locking-in-webkit/.

29 R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 116–123. IEEE Computer Society, 1990. doi:10.1109/ICDCS.1990.89257.

30 David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, February 1979. doi:10.1145/359060.359076.

31 Benoit Schillings. Be Engineering Insights: Benaphores. *Be Newsletters*, 1(26), May 1996.

32 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

33 Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 33–46. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Soares.pdf.

34 Roy Spliet, Manohar Vanga, Björn B. Brandenburg, and Sven Dziadek. Fast on average, predictable in the worst case: Exploring real-time futexes in LITMUSRT. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 96–105. IEEE Computer Society, 2014. doi:10.1109/RTSS.2014.33.

35 Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 45–56. IEEE Computer Society, 2013. doi:10.1109/RTSS.2013.13.

36 Alexander Zuepke. Deterministic fast user space synchronisation. In *OSPERT Workshop*, July 2013.

37 Alexander Zuepke, Marc Bommert, and Daniel Lohmann. AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 133–144. IEEE Computer Society, 2015. doi:10.1109/RTAS.2015.7108435.

38 Alexander Zuepke and Robert Kaiser. Deterministic futexes: Addressing WCET and bounded interference concerns. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 65–76. IEEE, 2019. doi:10.1109/RTAS.2019.00014.